Robot Programming Practice #3

Dept. of Mech. Robotics and Energy Eng.
Dongguk University

Timers and Interrupts

- Time and event management in embedded systems
- An introduction to timers
- Using the mbed Timer object
- Using multiple timers
- Using the mbed Ticker object
- Hardware interrupts
- External interrupts on the mbed
- Switch debouncing for interrupt control
- Extended exercises

Time and event management

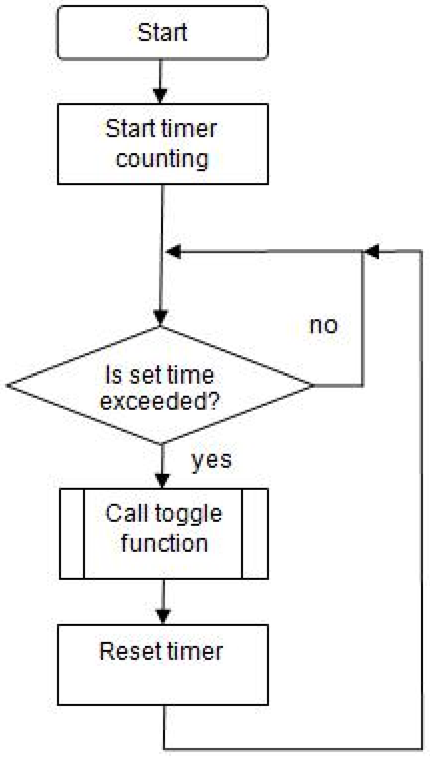
- Many embedded systems need high precision timing control and the ability to respond urgently to critical requests. For example:
  - A video camera needs to capture image data at very specific time intervals, and to a high degree of accuracy, to enable smooth playback
  - An automotive system needs to be able to respond rapidly to a crash detection sensor in order to activate the passenger airbag
- Interrupts allow software processes to be halted while another, higher priority section of software executes.
- Interrupt routines can be programmed to execute on timed events or by events that occur externally in hardware.

Time and event management

- Routines executed by events that occur from an external source (e.g. a mouse click or input from another program) can be referred to as ‘event driven’.
- Interrupts in embedded systems can be thought of as functions which are called by specific events rather than directly in code.
- The simplest type of interrupt is one which automatically increments a counter at a periodic interval, this is done behind the scenes while the software is operating.
- Most microcontrollers have built in timers or real-time-interrupts which can be used for this purpose.

A Simple Timer Routine

- Create a square wave output using scheduled programming.



```
#include "mbed.h"

Timer timer1;
DigitalOut myled(LED1);
void task1(void);

int main() {
    timer1.start();
    while(1) {
        if (timer1.read_ms()>=200)
        {
            task1();
            timer1.reset();
        }
    }
}

void task1(void) {
    myled = !myled;
}
```

Using Multiple Timers

- Add a second timer which will run at a different rate, you can use an LED or an oscilloscope on the mbed pins to check that the two timers are executing correctly.

```
#include "mbed.h"

Timer timer1;
Timer timer2;

DigitalOut myled1(LED1);
DigitalOut myled2(LED2);

void task1(void);
void task2(void);

int main() {
    timer1.start();
    timer2.start();
    while(1) {
        if (timer1.read_ms()>=200)
        {
```

```
                task1();
                timer1.reset();
        }
if (timer2.read_ms()>=500)
        {
            task2();
            timer2.reset();
        }
    }
}

void task1(void) {
    myled1 = !myled1;
}
void task2(void) {
    myled2 = !myled2;
}
```

Challenges with Timer Interrupts

- With scheduled programming, we should be careful with the amount of code and how long it takes to execute.
- For example, if we need to run a task every 1 ms, that task must take less than 1 ms second to execute, otherwise the timing would overrun and the system would go out of sync.
  - How much code there is will determine how fast the processor clock needs to be.
  - We sometimes need to prioritize the tasks, does a 1ms task run before a 100ms task? (because after 100ms, both will want to run at the same time).
  - This also means that pause, wait or delays (i.e. timing control by 'polling') cannot be used within scheduled program designs.

Using the mbed Ticker object

- The Ticker interface is used to setup a recurring interrupt to repeatedly call a designated function at a specified rate.
- Previously we used the Timer object, which required the main code to continuously analyse the timer to determine whether it was the right time to execute a specified function.
- An advantage of the Ticker object is that we don’t need to read the time, so we can execute other code while the ticker is running in the background and calling the attached function as necessary.

Using the mbed Ticker object

- Use two tickers to create square wave outputs.
- Use an LED or an oscilloscope on the mbed pins to check that the two tickers are executing correctly.

```
#include "mbed.h"
Ticker flipper1;
Ticker flipper2;

DigitalOut myled1(LED1);
DigitalOut myled2(LED2);

void flip1(){
    myled1 = !myled1;
}

void flip2(){
    myled2 = !myled2;
}
```

```
int main() {
    myled1 = 0;
    myled2 = 0;

    flipper1.attach(&flip1, 0.2);
    flipper2.attach(&flip2, 1.0);

    while(1) {
        wait(0.2);
    }
}
```

Hardware Interrupts

- Microprocessors can be set up to perform specific tasks when hardware events are incident.
- This allows the main code to run and perform its tasks, and only jump to certain subroutines or functions when something physical happens.
  - i.e. a switch is pressed or a signal input changes state.
- Interrupts are used to ensure adequate service response times in processing.
- The only real disadvantage of interrupt systems is the fact that programming and code structures are more detailed and complex.

External Interrupts on the mbed

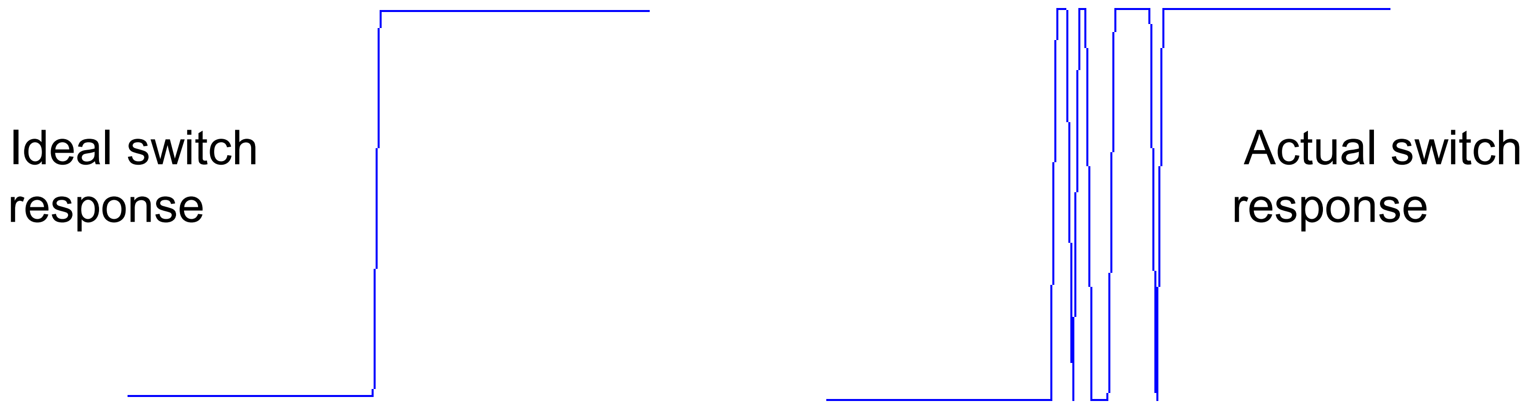
- Use the mbed InterruptIn library to toggle an LED whenever a digital pushbutton input goes high.

```
#include "mbed.h"

InterruptIn button(p5);  // Interrupt on p5
DigitalOut myled(LED1);   // digital out to LED1
void toggle(void);

int main() {
    button.rise(&toggle); // attach the address
}

void toggle() {
    myled = !myled;
}
```

- You may notice some issues with this simple program, what are they?

Switch debouncing for interrupt control

- Exercise doesn't work quite as expected; it is possible for the button to become unresponsive or out of synch with the LED.
- This is because of a common issue called switch or button bouncing. When the button is pressed it doesn't cleanly switch from low to high, there is some 'bounce' in between as shown below:



Switch debouncing for interrupt control

- It is therefore easy to see how a single button press can cause multiple interrupts and hence the LED can get out of synch with the button.
- We therefore need to ‘debounce’ the switch with a timer feature.

Switch debouncing for interrupt control

- Use the mbed InterruptIn library to toggle an LED whenever a digital input goes high, implementing a debounce counter to avoid multiple interrupts.

```
#include "mbed.h"

InterruptIn button(p5);   // Interrupt on p5
DigitalOut myled(LED1);    // digital out to LED1
Timer debounce;     // define debounce timer
void toggle(void);

int main() {
    debounce.start();
    button.rise(&toggle); // attach the address
}

void toggle() {
if (debounce.read_ms()>200)
    myled = !myled;
    debounce.reset();
}
```

Serial Communication

- The mbed Microcontroller can communicate with a host PC through a "USB Virtual Serial Port" over the same USB cable that is used for programming.
- This enables you to:
  - Print out messages to a host PC terminal (useful for debugging!)
  - Read input from the host PC keyboard
  - Communicate with applications and programming languages running on the host PC that can communicate with a serial port, e.g. perl, python, java and so on.

Serial Communication

- Your mbed Microcontroller can appear on your computer as a serial port. For Windows, you need to install a driver.
- Download the installer(mbedWinSerial_16466.exe) to your PC, e.g. your desktop.
- With your *mbed plugged in*, and *no explorer drive windows open*, run the installer:
- It will take some time, and pop up a few 'unsigned driver' warnings, but after a while you should have a Serial port.

Serial Communication

- It is common to use a *terminal application* on the host PC to communicate with the mbed Microcontroller.
- This allows the mbed Microcontroller to print to your PC screen, and for you to send characters back.
- Download and install a Terminal application(teraterm-4.77.exe).
- In this example we recommend the latest version of Teraterm:

Configure the connection

- Open and Setup Teraterm
  - File -> New Connection (or just press Alt+N)
  - Select the "Serial" radio button and choose the "mbed Serial Port" from the drop down menu.
  - Click "OK"
- Setup New-line format (to print out new line characters correctly)
  - Setup -> Terminal...
  - Under "New-line", set Receive to "LF"
- Your terminal program is now configured and connected.
- To change the baud rate, go to Setup -> Serial Port...

Test of USB comm between PC and mbed

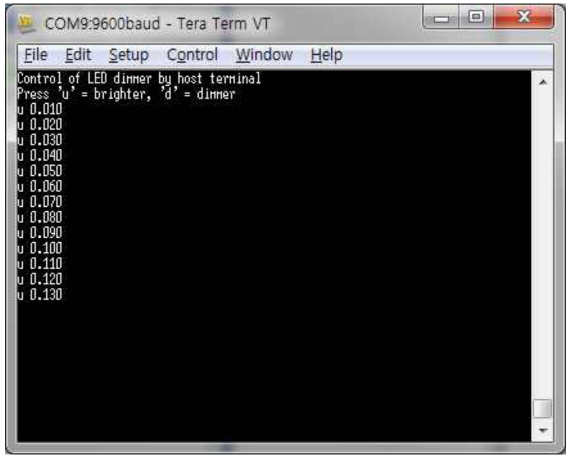
- Write the following program and download it. Use Teraterm to see the result.

```
#include "mbed.h"

Serial pc(USBTX, USBRX); // tx, rx

int main() {
    pc.printf("Hello World!\n");
    while(1) {
        pc.putc(pc.getc() + 1);
    }
}
```

Control of mbed from PC

- Connect to your mbed Microcontroller with a Terminal program and uses the 'u' and 'd' keys to make LED1 brighter or dimmer.

```
#include "mbed.h"

Serial pc(USBTX, USBRX); // tx, rx
PwmOut led(LED1);
float brightness = 0.0;

int main() {
    pc.printf("Control of LED dimmer by host terminal\n\r");
    pc.printf("Press 'u' = brighter, 'd' = dimmer\n\r");
    while(1) {
    char c = pc.getc();
    wait(0.001);
    if ((c=='u') && (brightness<0.5)) {
        brightness += 0.01;
        led = brightness;
        }
    if ((c=='d') && (brightness>0.0)) {
        brightness -= 0.01;
        led = brightness;
        }
        pc.printf("%c %6.3f \n\r",c,brightness);
    }
}
```


COM9:9600baud - Tera Term VT
File Edit Setup Control Window Help

```
Control of LED dimmer by host terminal
Press 'u' = brighter, 'd' = dimmer
u 0.010
u 0.020
u 0.030
u 0.040
u 0.050
u 0.060
u 0.070
u 0.080
u 0.090
u 0.100
u 0.110
u 0.120
u 0.130
```

Data Acquisition via USB

- Use the potentiometer(pin 20). Print out the AD value on the PC.

```
#include "mbed.h"

Serial pc(USBTX, USBRX); // tx, rx
AnalogIn Ain(p20);
float ADCdata;

int main() {
    pc.printf("ADC Data Values ... \n\r");
    while(1) {
        ADCdata = Ain;
        pc.printf("%f \n\r",ADCdata);
        wait(0.5);
    }
}
```

Callback (attach to RX interrup)

- Let’s control mbed by typing command from PC.

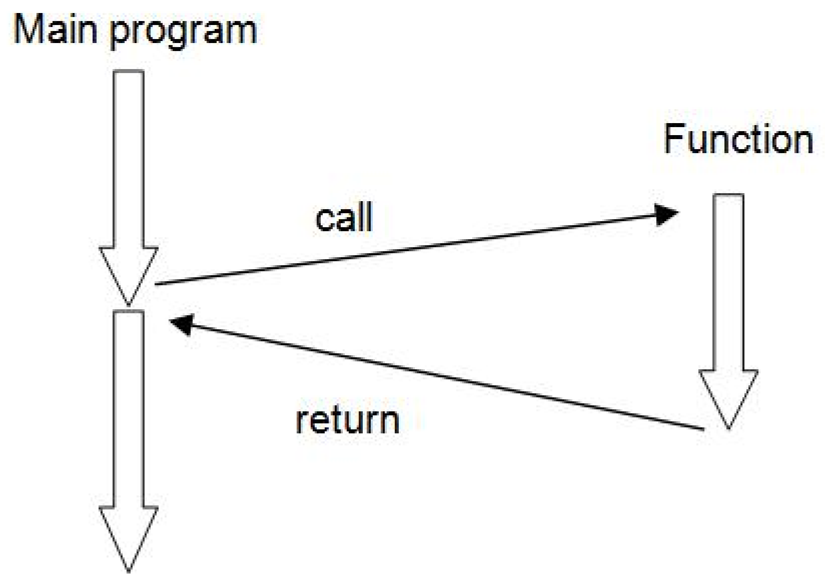
```
#include "mbed.h‟

DigitalOut led1(LED1);
DigitalOut led2(LED2);

Serial pc(USBTX, USBRX);

void callback() {
// Note: you need to actually read from the serial to clear the RX interrupt
 printf("%c\n", pc.getc());
  led2 = !led2;
}

int main() {
       pc.attach(&callback);
       while (1) {
         led1 = !led1;
         wait(0.5);
       }
}
```

Modular Programming

- When working on large, multi-functional projects it is particularly important to consider the design and structure of the software. It is not possible to program all functionality into a single loop!
- The techniques used in large multifunctional projects also bring benefits to individuals who work on a number of similar design projects and regularly reuse code.

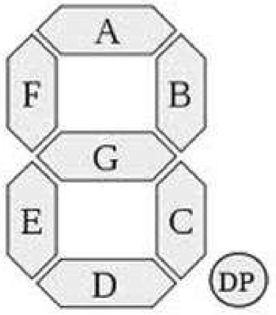
Functions and Subroutines

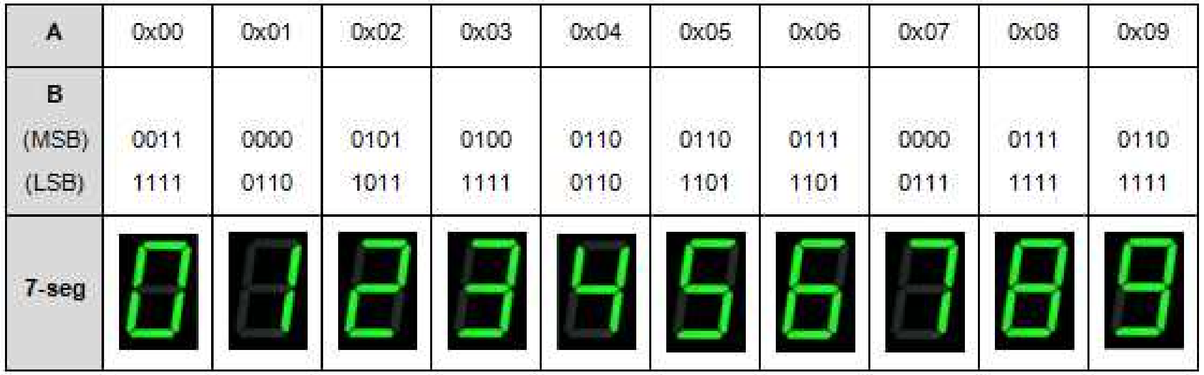
- A function (sometimes called a subroutine) is a portion of code within a larger program
- A function performs a specific task and is relatively independent of the main code



Functions and Subroutines

- To demonstrate the use of functions, we will look at examples using a standard 7-segment LED display.
- Note that the 8-bit byte for controlling the individual LED's on/off state is configured as: (MSB) DP G F E D C B A (LSB).

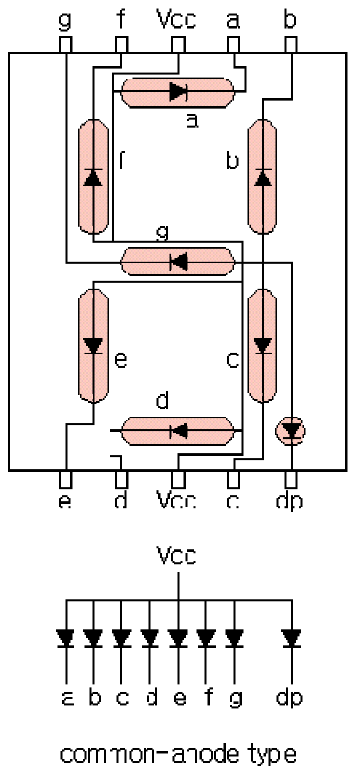


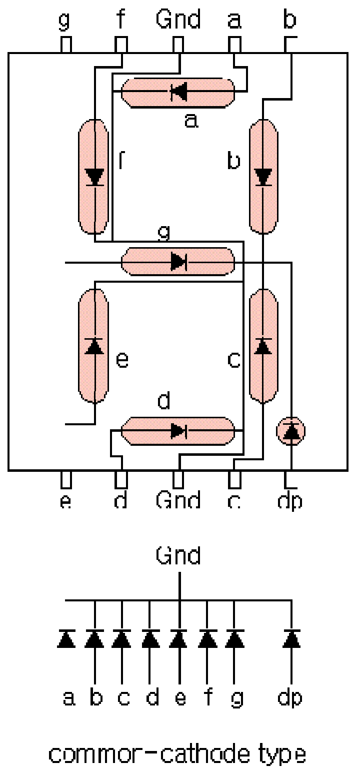


A	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09
B (MSB) (LSB)	0011 1111	0000 0110	0101 1011	0100 1111	0110 0110	0110 1101	0111 1101	0000 0111	0111 1111	0110 1111
7-seg	0	1	2	3	4	5	6	7	8	9

Working with 7-Segment Displays

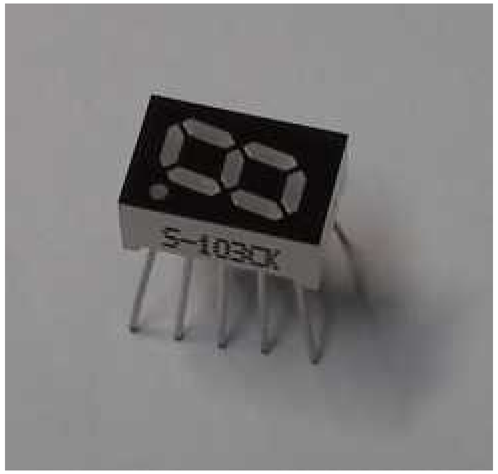
- There are Common-Anode type(Out type) and Common-Cathode Type.

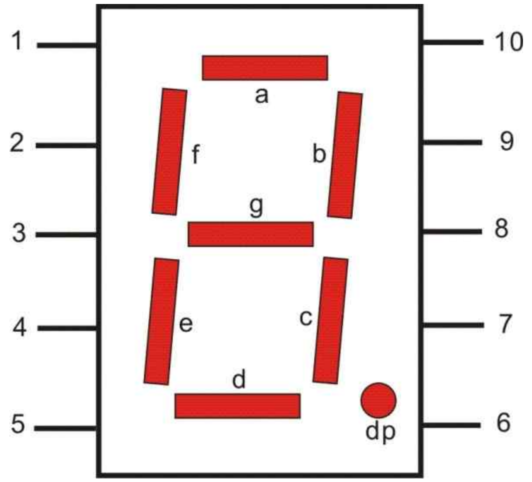


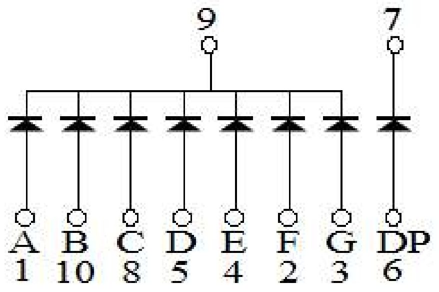


Working with 7-Segment Displays

- Common-Cathode Type.

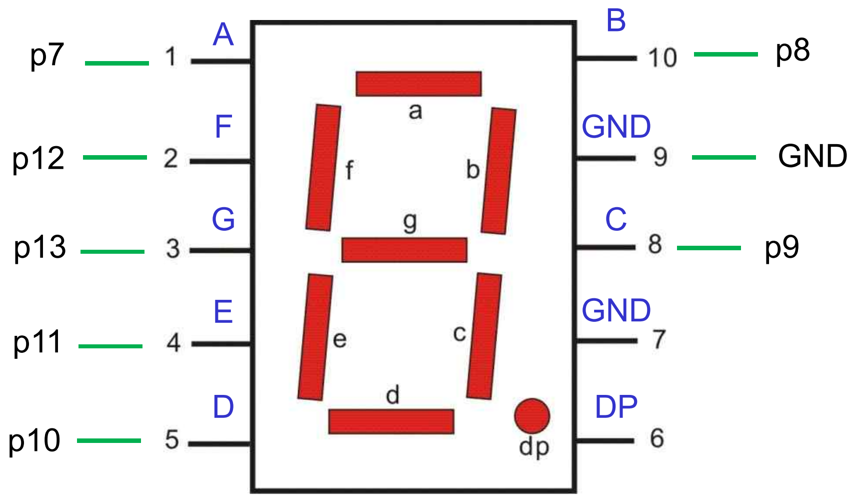


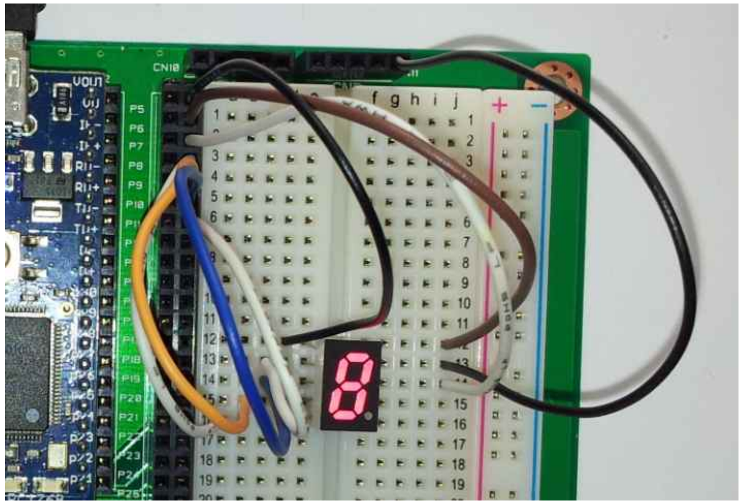




Working with 7-Segment Displays

- Connect A through G to pin 7 through pin 13.
- Pin 9 should be connected to GND.





Working with 7-Segment Displays

- A 7-segment display is actually just 8 LEDs in a single package. We can therefore connect each LED pin to an mbed pin to display a chosen number.

- For example, the code here uses a digital output for each LED segment and switches the correct LEDs on and off to display the number 3.

```
#include "mbed.h"
DigitalOut A(p7);
DigitalOut B(p8);
DigitalOut C(p9);
DigitalOut D(p10);
DigitalOut E(p11);
DigitalOut F(p12);
DigitalOut G(p13);
int main() {
A=1;
B=1;
C=1;
D=1;
E=1;
F=1;
G=1;
}
```

Working with 7-Segment Displays

- You may see that this code can get a little intensive; it is a very simple operation but not easy to represent number.

```
#include "mbed.h"
DigitalOut A(p7);
DigitalOut B(p8);
DigitalOut C(p9);
DigitalOut D(p10);
DigitalOut E(p11);
DigitalOut F(p12);
DigitalOut G(p13);
int main() {
A=1;
B=1;
C=1;
D=1;
E=0;
F=0;
G=1;
}
```

Working with 7-Segment Displays

- If we want our LED display to continuously count from 0-9, our code might look something like this

Working with 7-Segment Displays

```
#include "mbed.h"
DigitalOut A(p7);
DigitalOut B(p8);
DigitalOut C(p9);
DigitalOut D(p10);
DigitalOut E(p11);
DigitalOut F(p12);
DigitalOut G(p13);

int main() {
```

```
while (1) {
      A=1; B=1; C=1; D=1; E=1; F=1; G=0;      // '0'
      wait(0.2);
      A=0; B=1; C=1; D=0; E=0; F=0; G=0;      // '1'
      wait(0.2);
      A=1; B=1; C=0; D=1; E=1; F=0; G=1;      // '2'
      wait(0.2);
      A=1; B=1; C=1; D=1; E=0; F=0; G=1;      // '3'
      wait(0.2);
      A=0; B=1; C=1; D=0; E=0; F=1; G=1;      // '4'
      wait(0.2);
      A=1; B=0; C=1; D=1; E=0; F=1; G=1;      // '5'
      wait(0.2);
      A=1; B=0; C=1; D=1; E=1; F=1; G=1;      // '6'
      wait(0.2);
      A=1; B=1; C=1; D=0; E=0; F=0; G=0;      // '7'
      wait(0.2);
      A=1; B=1; C=1; D=1; E=1; F=1; G=1;      // '8'
      wait(0.2);
      A=1; B=1; C=1; D=1; E=0; F=1; G=1;      // '9'
      wait(0.2);
  }
}
```

Working with 7-Segment Displays

- Before moving on to functions, we can simplify our code using a BusOut object.
- The BusOut object allows a number of digital outputs to be configured and manipulated together.
- We can therefore define a BusOut object for our 7-segment display and send the desired data byte value in order to display a chosen number.
- Verify that the code here performs the same functionality as that in the previous exercise.

Working with 7-Segment Displays

- Using BusOut and Hexadecimal number, we can rewrite the previous program.

```
#include "mbed.h"

BusOut Seg1(p7,p8,p9,p10,p11,p12,p13,p14); //ABCDEFGDP

int main() {
while (1) {                                    //DPGFEDCBA
      Seg1 = 0x3F;          wait(0.2);   // 00111111
      Seg1 = 0x06;          wait(0.2);   // 00000110
      Seg1 = 0x5B;          wait(0.2);   // 01011011
      Seg1 = 0x4F;          wait(0.2);   // 01001111
      Seg1 = 0x66;          wait(0.2);   // 01100110
      Seg1 = 0x6D;          wait(0.2);   // 01101101
      Seg1 = 0x7D;          wait(0.2);   // 01111101
      Seg1 = 0x07;          wait(0.2);   // 00000111
      Seg1 = 0x7F;          wait(0.2);   // 01111111
      Seg1 = 0x6F;          wait(0.2);   // 01101111
  }
}
```

Designing a C function

- It is beneficial for us to design a C function that inputs a count variable and returns the 8-bit value for the corresponding 7-segment display, for example:

```
char SegConvert(char SegValue)  {
      char SegByte = 0x00;
      switch (SegValue) {
         case 0 : SegByte = 0x3F;        break;
         case 1 : SegByte = 0x06;        break;
         case 2 : SegByte = 0x5B;        break;
         case 3 : SegByte = 0x4F;        break;
         case 4 : SegByte = 0x66;        break;
         case 5 : SegByte = 0x6D;        break;
         case 6 : SegByte = 0x7D;        break;
         case 7 : SegByte = 0x07;        break;
         case 8 : SegByte = 0x7F;        break;
         case 9 : SegByte = 0x6F;        break;
       }
    return SegByte;
}
```

- Note that this function uses a C ‘switch/case’ statement which performs like an ‘If’ operation with a number of possible conditions.

Implementing a C function

- Verify that the following program, using the SegConvert function, performs the same functionality as in the previous exercises.

Implementing a C function

```
#include "mbed.h"

BusOut Seg1(p7,p8,p9,p10,p11,p12,p13,p14); //ABCDEFGDP

char SegConvert(char SegValue);

int main() {
while (1) {
      for (char i=0; i<10; i++) {
          Seg1 = SegConvert(i);
          wait(0.2);
      }
   }
}

char SegConvert(char SegValue)  {
      char SegByte = 0x00;
      switch (SegValue) {
         case 0 : SegByte = 0x3F;        break;
         case 1 : SegByte = 0x06;        break;
         case 2 : SegByte = 0x5B;        break;
         case 3 : SegByte = 0x4F;        break;
         case 4 : SegByte = 0x66;        break;
         case 5 : SegByte = 0x6D;        break;
         case 6 : SegByte = 0x7D;        break;
         case 7 : SegByte = 0x07;        break;
         case 8 : SegByte = 0x7F;        break;
         case 9 : SegByte = 0x6F;        break;
       }
    return SegByte;
}
```